

Verifying χ models in SPIN

Nikola Trčka

`n.trcka@tue.nl`

Eindhoven University of Technology
Faculty of Mathematics and Computer Science,
Formal Methods Group

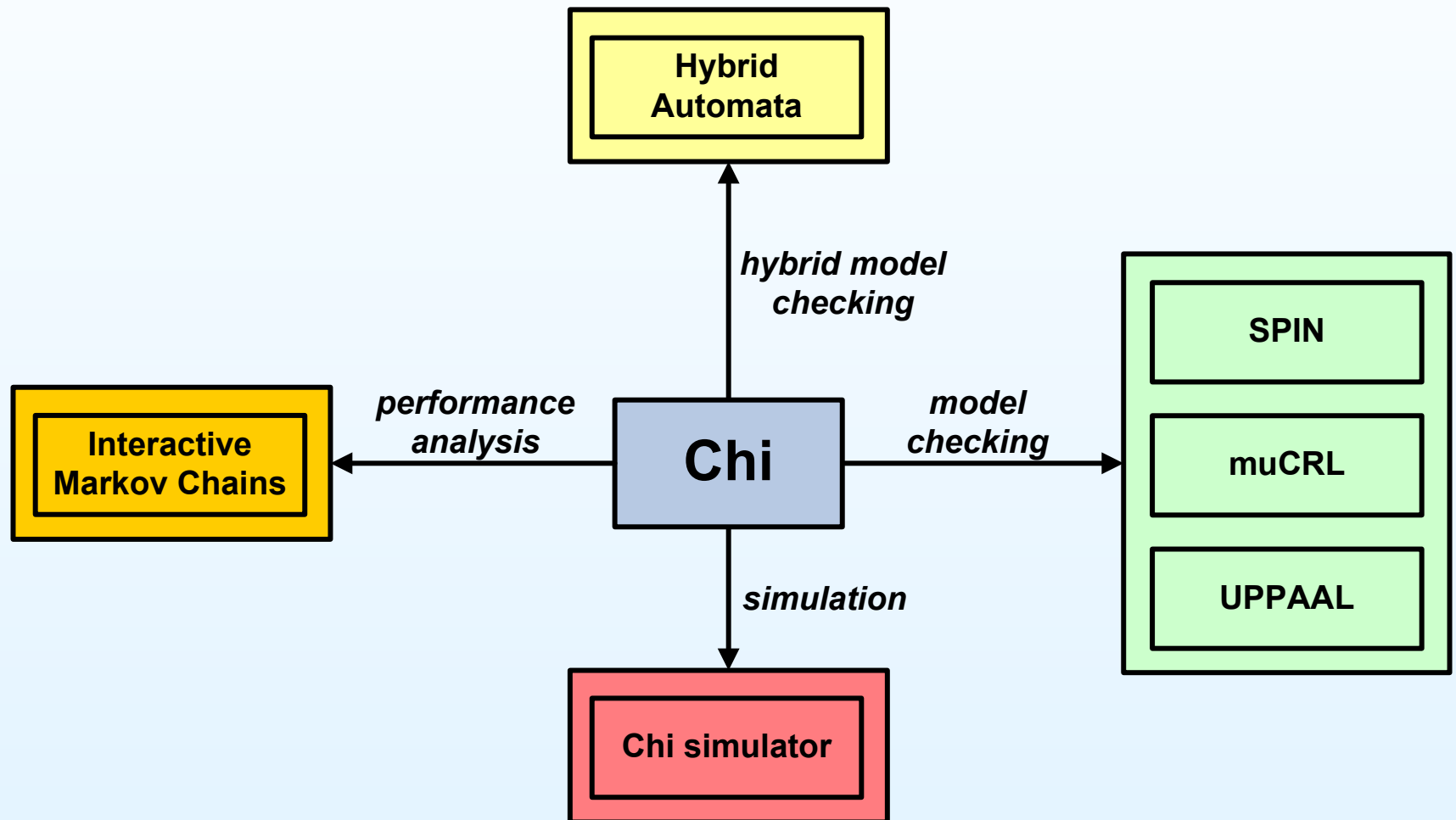
What is χ ?

- language for modeling, simulation and control of manufacturing systems (machines, production cells, factories ...)
- developed by Systems Engineering Group, Faculty of Mechanical Engineering, TU/e
- process algebra like
- data types
- continuous timing
- discrete-event and hybrid models
- many industrial cases

How to verify χ models?

- Model checking most popular
- Two ways:
 1. build a model checker for χ
 - hard to beat the existing ones
 - must choose temporal logic (CTL/LTL, action/state based)
 - etc.
 2. translate a model to an input language of a popular model checker
 - case study: turntable machine
 - model translated to three different formalisms
 - verification in all three environments
 3. currently the second option is pursued

χ Analysis Environment



λ Data Types

- Basic:
 - channel
 - boolean
 - natural, integer and rational numbers
- Compound:
 - lists
 - $l : [int] \mapsto []$
 - $++$, hd , tl , len , ...
 - tuples
 - $t : \langle int, nat \rangle \mapsto \langle -2, 3 \rangle$
 - $t.i$
 - sets

χ - Syntax

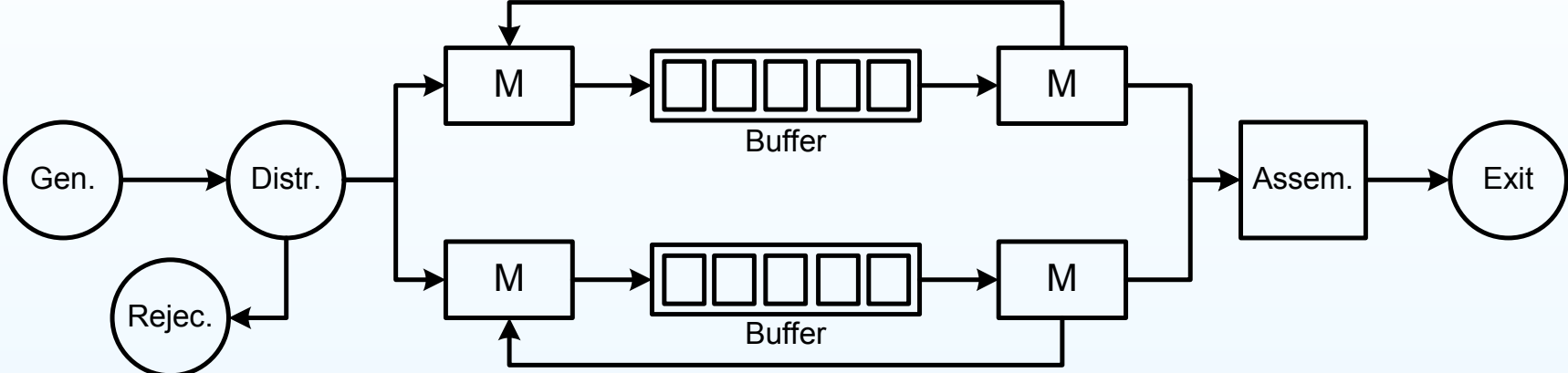
$A ::= skip \mid x_1, \dots, x_n := e_1, \dots, e_n \mid m!e \mid m?x \mid m!!e \mid m$

$P ::=$

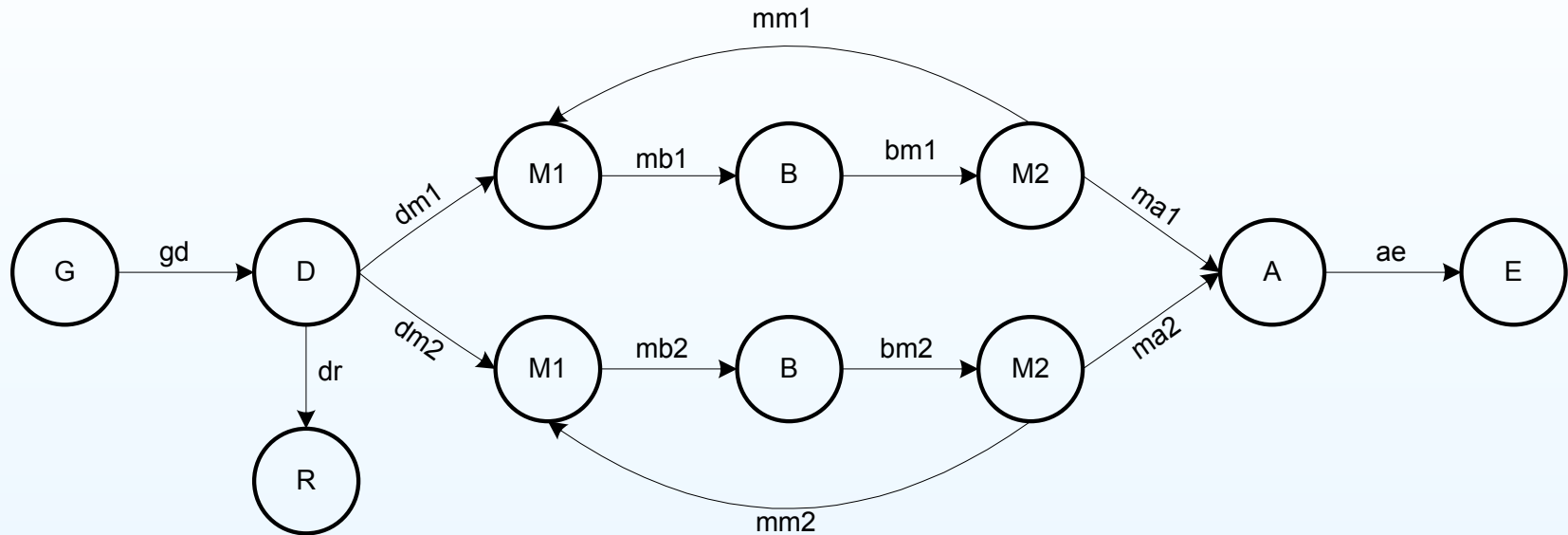
- A
- $b \rightarrow P$
- $P ; P$
- $P \square P$
- $*P$
- $b \xrightarrow{*} P$
- $P \parallel P$
- $[[s \mid P]]$
- $\partial_A(P)$
- $\mathcal{U}_{\mathcal{H}}(P)$

$S ::= discx : type = c, S' \mid chanx : type, S' \mid l : [type] = l', S'$

Example: Manufacturing line



Manufacturing line in χ



$\mathbf{B}(\text{chan } in, out : \text{bool}, \text{disc } n : \text{int}) = \llbracket [\text{disc } x : \text{bool}, buf : [\text{bool}] = [] \mid$

$\ast(\text{len}(buf) < n \rightarrow (in?x ; buf := buf ++ [x])$

$\llbracket \text{len}(buf) > 0 \rightarrow (out!hd(buf) ; buf := tl(buf)) \rrbracket$

$\mathbf{A}(\text{chan } in_1, in_2, out : \text{bool}, \text{disc } d : \text{int}) = \llbracket [\text{disc } x, y : \text{bool} \mid$

$\ast((in_1?x \parallel in_2?y) ; \Delta d ; out!(x \wedge y)) \rrbracket$

$\text{sys}() = \mathcal{U}_{\mathcal{H}} \partial_{\mathcal{A}} (\llbracket \text{chan } gd, \dots, ae : \text{bool} \mid \mathbf{G}(gd, 7) \parallel \dots \parallel \mathbf{B}(mb_1, bm_1, 5) \parallel \mathbf{B}(mb_2, bm_2, 5) \parallel \mathbf{A} \rrbracket$

Introduction to SPIN / PROMELA [Holzmann]

1. What is SPIN / PROMELA?
2. Data Types in PROMELA
3. Features of the language
4. PROMELA Model - Structure
5. Example: Sleep/Wake Up Scheduling
6. DTPROMELA: Adding Time to PROMELA

What is SPIN / PROMELA?

PROMELA: *Protocol Meta Language*

- specification language to model finite-state concurrent systems
- syntax resembles that of the programming language C
- communication features from CSP

SPIN: *Simple Promela Interpreter*

- tool to analyze systems modeled in PROMELA
- simulator
- verifier (on the fly, state-based LTL model checking)

XSPIN: Graphical user interface for SPIN, Tcl/Tk Code

PROMELA - Data Types

- Basic

- `bool`
- `byte`, `short`, `int` (all with the unsigned possibility)

- **channels**

```
chan m=[2] of {byte}
```

```
chan m=[0] of {int, bool}
```

- Mechanisms to build new types

- `mtype`
- `typedef`

PROMELA - Features of the Language

- assignments (`x := Expr`)
- alternative composition (`if :: fi`)
- sequential composition (`;`)
- repetition (`do :: od`), exit by explicit `break` or `goto`
- `atomic` and `d_step` reduce a state space
- expressions are executable
- implicit parallelism

PROMELA Model - Structure

A PROMELA model consists of:

- type definition
- global channel declarations
- global variable declarations
- process declarations
 - local variables and channels
 - process body
- `init` process
 - start the execution of processes

DTPROMELA [Bosnacki]

- Discrete time extension
- Set of decreasing timers
- Synchronization on `timeout` action
- Time factorization
- Maximal progress

```
active proctype Sender(){
    timer t;

    delay(t,2);
    if
        :: m!0
        :: skip
    fi
}
```

Note: `delay(t,2)` = `set(t,2);expire(t)` = `t.val=2;t.val==0`

Translating χ to PROMELA - Overview

1. Assumptions
2. Straightforward Translations
3. Translation of:
 - data types
 - nested scoping
 - immediate termination and repetition
 - delays
 - guarded processes
 - nested parallelism
4. Summary - Translatable Subset and Translation Process
5. Example: The Manufacturing Line in PROMELA

Translation Assumptions

We only consider:

$$\mathcal{U}_{\mathcal{H}}\partial_A(p)$$

- \mathcal{H} is the set of all channels used
- A is the set of all send and receive actions
- p does not contain ∂ nor $\mathcal{U}_{\mathcal{H}}$.

Reasons:

- main form of communication in χ is synchronous
- there is no encapsulation in PROMELA
- communication urgency in SPIN is implicit

Straightforward Translations

- $\delta \longmapsto \text{false}$
- $\text{skip} \longmapsto \text{skip}$
- $p \parallel q \longmapsto \begin{array}{l} \text{if} \\ \quad :: p \\ \quad :: q \\ \text{fi} \end{array}$
- $p ; q \longmapsto \begin{array}{l} p ; \\ q \end{array}$
- $*p \longmapsto \begin{array}{l} \text{do} \\ \quad :: p \\ \text{od} \end{array}$

Straightforward Translation cont.

$m!!e \mapsto$

```
if
  :: m!e
  :: atomic {timeout; false}
fi
```

$m??x \mapsto$

```
if
  :: m?x
  :: atomic {timeout; false}
fi
```

$b \xrightarrow{*} p \mapsto$

```
do
  :: b; p
  :: !b
od
```

Translation - Data Types

- *bool*, *nat* and *int* straightforward
- no rational numbers in PROMELA
- channels of length 0

Example:

$x : nat \mapsto 3, \sim m \quad \mapsto$ `unsigned int x = 3;
chan m = [0] of {int}`

Translation - Tuples

- New type

```
typedef type1_type2 {  
    type1 elem1;  
    type2 elem2  
}
```

$\llbracket t : \langle int, bool \rangle \mid \dots t.1 := 5 \dots \rrbracket \mapsto$

```
int_bool t;
```

```
\dots t.elem1 = 5 \dots
```

Translation - Lists

- using channels of size n

```
typedef LIST_type {
    chan l = [n] of {type};
    type head;
}
```

- operators on lists

```
#define add(x,lst) d_step{ lst.l!x;
                        if
                            :: len(lst.l) == 1 -> lst.head = x
                            :: else
                        fi }

#define hd(lst) (lst.head)
#define tail(lst) d_step{ lst.l?_;
                        if
                            :: len(lst.l) > 0 -> lst.l?<lst.head>
                            :: else
                        fi }

#define length(lst) (len(lst.l))
```

Translation - Nested Scopes

- recall only two scope levels in PROMELA
- nested scopes can be eliminated after some renaming

$$b \rightarrow \llbracket s \mid p \rrbracket \longmapsto \llbracket s \mid b \rightarrow p \rrbracket$$

- more complicated in case of repetition

$$*\llbracket s \mid p \rrbracket \longmapsto \llbracket s \mid *(p ; x_1, \dots, x_n \dots := c_1, \dots, c_n) \rrbracket$$

$$b \xrightarrow{*} \llbracket s \mid p \rrbracket \longmapsto \llbracket s \mid b \xrightarrow{*} (p ; x_1, \dots, x_n \dots := c_1, \dots, c_n) \rrbracket$$

assuming $s = x : type_1 = c_1, \dots, x : type_n = c_n$.

Translation - Timing

$\Delta e \mapsto \text{delay}(t, e)$

no deadlock

$\Delta 2; \delta \parallel \Delta 2; p$

possible deadlock

```
if
  :: delay(t1, 2); false
  :: delay(t2, 2); p
fi
```

Reason:

- recall that $\text{delay}(t, 2) = \text{set}(t, 2); \text{expire}(t)$
- $\text{set}(t, e)$ can make a choice

Solution: $\text{set}(t, e)$ must be moved to some place "safe"

```
 $\Delta 2; \delta \parallel \Delta 2; p \mapsto$ 
  set(t1, 2); set(t2, 2);
  if
    :: expire(t1); false
    :: expire(t2); p
  fi
```

Guards (naive translation)

$$b \rightarrow p \quad \longmapsto \quad b \text{ -> } p$$

Guards (naive translation)

$$b \rightarrow p \not\mapsto b \rightarrow p$$

Reasons:

- in χ , process $b \rightarrow p$ continues if b is true and p can execute an action
- boolean expressions in PROMELA are also statements
- $b \rightarrow p$ is equivalent to $b; p$

no deadlock

$(true \rightarrow \delta \mid true \rightarrow skip)$

possible deadlock

$\not\mapsto$

```
if
  :: true -> false
  :: true -> skip
fi;
```

Guards (correct translation)

- only guarded atomic processes can be translated
- additional field in channel declarations

$$b \rightarrow x := e \quad \longmapsto \quad \text{d_step}\{b; x := e\}$$

$$b \rightarrow \text{skip} \quad \longmapsto \quad b$$

$$b \rightarrow m!e \quad \longmapsto \quad m!e, b$$

$$b \rightarrow m?x \quad \longmapsto \quad m?x, \text{eval}(2-b)$$

$$b \rightarrow (\Delta e; \text{skip}) \quad \longmapsto \quad \text{set}(t, e); b \ \&\& \ \text{expire}(t)$$

Simplification of Guards

- almost all processes can be simplified to have only atomic subprocesses guarded

$b_1 \rightarrow b_2 \rightarrow p$	$(b_1 \wedge b_2) \rightarrow p$
$b \rightarrow (p \mid q)$	$(b \rightarrow p) \mid (b \rightarrow q)$
$b \rightarrow (p; q)$	$(b \rightarrow p) ; q$
$b \rightarrow p^*$	$(b \rightarrow p) ; p^* \mid b \rightarrow \varepsilon$

- $b \rightarrow (p \parallel q)$ is the same as $(b \rightarrow p) \parallel (b \rightarrow q)$ only when p and q do not change the value of b

Translation - Nested Parallelism

- recall that there is only implicit parallelism in PROMELA
- nested parallelism cannot in general be translated
- in some cases, it can be eliminated:
 - when it represents an arbitrary order of actions

$$a?x \parallel b?x \quad \longmapsto \quad (a?x ; b?x) \parallel (b?x ; a?x)$$

- in the sequential operator context

$$(p \parallel q) ; r \quad \longmapsto \quad \llbracket w : nat = 0 \mid p ; w := w + 1 \parallel q ; w := w + 1 \parallel w = 2 \rightarrow r \rrbracket$$

$$p ; (q \parallel r) \quad \longmapsto \quad \llbracket w : bool = false \mid p ; w := true \parallel w \rightarrow q \parallel w \rightarrow r \rrbracket$$

Future work

- find more reductions that are correct modulo stuttering congruence
- extend the stuttering congruence to cover the continuous time support of χ
- investigate further applications in domains outside χ